CSE 105
THEORY OF COMPUTATION

"Winter" 2018

http://cseweb.ucsd.edu/classes/wi18/cse105-ab/

Today's learning goals

Sipser Ch 5.1

- Define and explain core examples of computational problems, include A_{**}, E_{**}, EQ_{**}, $HALT_{TM}$ (for ** either DFA or TM)
- Explain what it means for one problem to reduce to another
- Use reductions to prove undecidability (or decidability)

A_{TM}

Sipser p. 207

A_{TM} = {<M,w> | M is a TM and w is in L(M) }

Define the TM N = "On input <M,w>:

1. Simulate M on w.
2. If M accepts, accept. If M rejects, reject."

loops

N is a Turing machine that recognizes A_{TM}.

No Turing machine decides A_{TM}.

ATM is undecidable.

A_{TM}

Sipser p. 210

- Recognizable
- Not decidable

$A_{TM}^{c} = \{\langle M, w\rangle \mid M$ does not accept w or $\langle M, w\rangle$ is not a valid input$\}$

not decidable.

is this recognizable? No

A_{TM}

Sipser p. 210

- Recognizable
- Not decidable

$\Rightarrow$ $\boxed{A_{TM}^{c} \text{ is not recognizable.}}$

Fact (from discussion section): A language is decidable iff it and its complement are both recgonizable.

Corollary 4.23: The complement of A_{TM} is **unrecognizable**.

Decidable	Recognizable (and not decidable)	Co-recognizable (and not decidable)
A_{DFA}	A_{TM}	$A_{TM}{}^{C}$
E_{DFA}		
EQ_{DFA}		

Give algorithm!

Diagonalization

Idea

Sipser pp. 215-216

If problem X is no harder than problem Y

…and if Y is easy

…then X must also be easy

Idea

Sipser pp. 215-216

If problem X is no harder than problem Y

…and if X is hard

…then Y must also be hard

Idea

Sipser pp. 215-216

If problem X is no harder than problem Y

…and if Y is **decidable**

…then X must also be **decidable**

If problem X is no harder than problem Y

…and if X is **undecidable**

…then Y must also be **undecidable**

Idea

Sipser pp. 215-216

If problem X is no harder than problem Y
...and if Y is **decidable**
...then X must also be **decidable**

If problem X is no harder than problem Y
...and if X is **undecidable**
...then Y must also be **undecidable**

"Problem X is no harder than problem Y" means
"Given information about Y, we could solve problem X".

The halting problem!

$\mathbf{HALT_{TM}}$ = { <M,w> | M is a TM and M halts on input w}

A_{TM} = { <M,w> | M is a TM and w is in L(M)}

How is $HALT_{TM}$ related to A_{TM} ?

A. They're the same set.
B. $HALT_{TM}$ is a subset of A_{TM}
C. A_{TM} is a subset of $HALT_{TM}$
D. They have the same type of elements but no other relation.
E. I don't know.

The halting problem!

$$\mathbf{HALT_{TM}} = \{ \langle M,w \rangle \mid M \text{ is a TM and } M \text{ halts on input } w\}$$

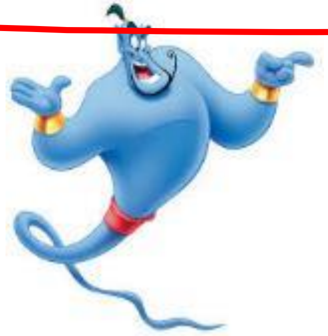
$$A_{TM} = \{ \langle M,w \rangle \mid M \text{ is a TM and } w \text{ is in } L(M)\}$$

But subset inclusion doesn't determine difficulty!

Claim: A_{TM} is no harder than $HALT_{TM}$

In other words: we could use $HALT_{TM}$ to solve A_{TM}

Given: Turing machine M, string w, magic genie for $HALT_{TM}$



Goal: Accept if w is in L(M); Reject if w is not in L(M).

Claim: A_{TM} is no harder than $HALT_{TM}$

In other words: we could use $HALT_{TM}$ to solve A_{TM}

Given: Turing machine M, string w, magic genie for $HALT_{TM}$

We can ask the magic genie "does a certain TM halt on certain input string?" Genie will **magically** give correct yes/no answer.

We can ask the genie as many of these questions as we'd like, about *any* TM and *any* string.

Goal: Accept if w is in L(M); Reject if w is not in L(M).

Claim: A_{TM} is no harder than $HALT_{TM}$

In other words: we could use $HALT_{TM}$ to solve A_{TM}

Given: Turing machine M, string w, magic genie for $HALT_{TM}$

"On input <M,w>

1. Ask Genie about M and w.
2. If Genie says no, then reject; if Genie says yes, run M on w.
   a. If this computation accepts, accept.
   b. If this computation rejects, reject."

Goal: Accept if w is in L(M); Reject if w is not in L(M) ?

Reduction

"Problem X **reduces to** problem Y" means

"Problem X is no harder than problem Y" means

"Given a genie for problem Y, we could solve problem X" means

"Given a solution for Y, we have a solution for X"

In the previous example, we used a genie for $HALT_{TM}$ to solve A_{TM}.

*Thus, **A_{TM} reduces to $HALT_{TM}$***

Which is **not** true?

A. ~~$HALT_{TM}$ reduces to A_{TM}~~

B. ~~Σ^* reduces to A_{TM}~~ True

C. A_{TM} reduces to {} (the empty set) not True

D. More than one of the above

E. None of the above

A_{TM} reduces to $Halt_{TM}$.

$\Rightarrow$ A_{TM} is no harder than $Halt_{TM}$

$\Rightarrow$ A_{TM} is undecidable implies $Halt_{TM}$ is undecidable.

Using reduction to prove undecidability

Claim: Problem X is undecidable

Proof strategy: Show that A_{TM} reduces to X.

Alternate Proof strategy: Show that $HALT_{TM}$ reduces to X.

etc.

In each of these, have access to Genie which can answer questions about X.

SCOOPING THE LOOP SNOOPER

A proof that the Halting Problem is undecidable

Geoffrey K. Pullum

(http://www.lel.ed.ac.uk/~gpullum/loopsnoop.html)

No general procedure for bug checks will do.
Now, I won't just assert that, I'll prove it to you.
I will prove that although you might work till you drop,
you cannot tell if computation will stop.

For imagine we have a procedure called P
that for specified input permits you to see
whether specified source code, with all of its faults,
defines a routine that eventually halts.

You feed in your program, with suitable data,
and P gets to work, and a little while later
(in finite compute time) correctly infers
whether infinite looping behavior occurs.

If there will be no looping, then P prints out 'Good.'
That means work on this input will halt, as it should.
But if it detects an unstoppable loop,
then P reports 'Bad!' — which means you're in the soup.

Well, the truth is that P cannot possibly be,
because if you wrote it and gave it to me,
I could use it to set up a logical bind
that would shatter your reason and scramble your mind.

Here's the trick that I'll use — and it's simple to do.
I'll define a procedure, which I will call Q,
that will use P's predictions of halting success
to stir up a terrible logical mess.

For a specified program, say A, one supplies,
the first step of this program called Q I devise
is to find out from P what's the right thing to say
of the looping behavior of A run on A.

If P's answer is 'Bad!', Q will suddenly stop.
But otherwise, Q will go back to the top,
and start off again, looping endlessly back,
till the universe dies and turns frozen and black.

And this program called Q wouldn't stay on the shelf;
I would ask it to forecast its run on itself.
When it reads its own source code, just what will it do?
What's the looping behavior of Q run on Q?

If P warns of infinite loops, Q will quit;
yet P is supposed to speak truly of it!
And if Q's going to quit, then P should say 'Good.'
Which makes Q start to loop! (P denied that it would.)

No matter how P might perform, Q will scoop it:
Q uses P's output to make P look stupid.
Whatever P says, it cannot predict Q:
P is right when it's wrong, and is false when it's true!

I've created a paradox, neat as can be —
and simply by using your putative P.
When you posited P you stepped into a snare;
Your assumption has led you right into my lair.

So where can this argument possibly go?
I don't have to tell you; I'm sure you must know.
A reductio: There cannot possibly be
a procedure that acts like the mythical P.

You can never find general mechanical means
for predicting the acts of computing machines;
it's something that cannot be done. So we users
must find our own bugs. Our computers are losers!

Next time

Pre-class reading for Wednesday Theorem 5.2